Rechnerarchitektur I & II

Stellenwertsysteme. **Dezimal → Dual:** sukzessives dividieren durch 2 und Reste notieren (z.B.: $10_{10}/2_{10} = 5_{10}|0_{10}$, $5_{10}/2_{10} = 2_{10}|1_{10}$, $2_{10}/2_{10} = 1_{10}|0_{10}$, $1_{10}/2_{10} = 0_{10}|1_{10}$, also $10_{10} = 0101_2$), bei Nachkommastellen Multiplizieren statt dividieren; **Dezimal → Oktal/Hexadezimal:** erst in Dual und dann in Endformat; **Dual → Dezimal:** Zweierpotenzen addieren; **Hexadezimalnotation:** z.B.:`0xAABBCC`= $AABBCC_{16}$

Flags. **Overflow** (Überlauf) (1 falls Überlauf stattfand, d.h. Addition 2er positiver/negativer Zahlen und Vorzeichen ändert sich (z.B.: 1000 + 1000 = 0000 mit Übertrag 1 also Overflow auch 1), Carry (Übertrag) (Carry des MSB (most significant bit)), Zero (ist null?), Sign (Vorzeichen), Parity (Parität, d.h. Summe aller bits); alle Flags werden so gesetzt, als ob die Zahl als Vorzeichenbehaftet interpretiert wird;

Komplementdarstellungen: *B*-**Komplement (2er-Kompl.):** $z_n \cdots z_{0_b} \mapsto (b-1-z_n)\ldots(b-1-z_0)_b + 1$ (n …Stellenanzahl); **1en-Komplement:** $z_n \cdots z_{02} \mapsto (1-z_n)\cdots(1-z_0)$;

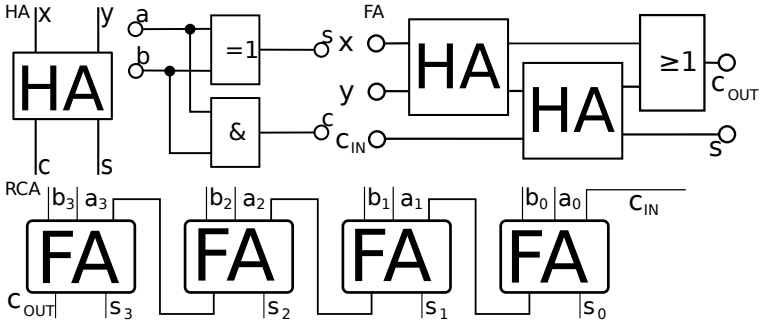
Standard IEEE754 - Single. $\boxed{b_{31}\cdots b_0}$, mit *Vorzeichen* $s = b_{31}$, *Charakteristik* $c = b_{30}\cdots b_{23} = B + e$ ($B = 127$ Bias, e Exponent), *fractional part* f (Mantisse) $b_{22}\cdots b_0 = 1, f$ (normalisiert, d.h. $0 < c < 2B+1$, $Z = (-1)^s \cdot (0,f) \cdot 2^{c-B}$) oder $= 0, f$ (denormalisiert, d.h. $c = 0$, $Z = (-1)^s \cdot (0,f) \cdot 2^{1-B}$, *unendlich* ($\pm\infty$) bei $c = 2B+1$, $f = 0$, *NaN* bei $c = 2B+1$, $f \neq 0$;

Stack. meist nach unten wachsender Stapel; besitzt Stackpointer (auf orberstes Element/'leere Zelle darüber') & Framepointer (auf letzte Rücksprungadresse;

Logikschaltungen. **Operatoren:** *and:* $\wedge, \cdot$, *or:* $\vee, +$, *xor:* $\oplus$; **KNF, KKNF, DNF, KDNF:** *KNF:* Konjunktion von Disjunktionen von Literalen, *DNF:* Disjunktion von Konjunktionen von Literalen, *kanonisch:* jede Variable kommt in jedem MAX/MIN-Term als Literal vor; **KV-Diagramme:** *Strategie:* große Rechtecke, *Unbestimmte Spezifikation:* $*$, z.B.:

	x_1x_2	$x_1\overline{x}_2$	$\overline{x}_1\overline{x}_2$	$\overline{x}_1x_2$
x_0	1	0	$*$	1
$\overline{x}_0$	1	0	$*$	1

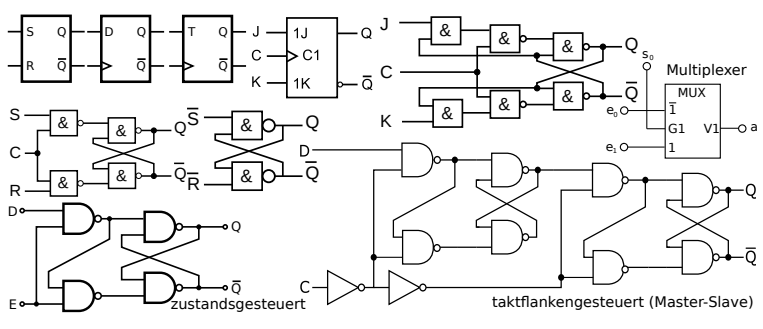
ergibt $x_1x_2 + \overline{x}_1$. **Halbaddierer (HA):** *Übertrag:* $c_{\text{out}} = ab$, *Summe:* $s = a \oplus b$. **Volladdierer (FA):** *Übertrag:* $c_{\text{out}} = c_{\text{in}}(a \oplus b) + ab$, *Summe:* $s = c_{\text{in}} \oplus a \oplus b$, **Ripple-Carry-Addierer:** mehrere Volladdierer hintereinander (mehrstellige Zahlen), **Subtrahierer:** Addierer mit $a, \overline{b}$ und $c_{-1} = 1$ (ergibt Resultat im 2er Komplement) *Schaltungen:*



Flip-Flops: *Arten:* RS-FF (Setzen-Rücksetzen), D-FF (Data-Clock), T-FF (Toggle-Clock, lässt sich aus D-FF durch Rückkopplung des negierten Ausgangs auf Eingang bauen), JK-FF (Jump-Kill-Toggle, immer flankengesteuert oder als Master-Slave-FF), *Tabellen:*

$R\,S$	Q^+_{RS}	Q_D	Q^+_T	$J\,K$	Q^+_{JK}
0 0	bleibt	D	$\overline{Q}_T$	0 0	Q_{JK}
0 1	1			0 1	0
1 0	0			1 0	1
1 1	instabil			1 1	$\overline{Q}_{JK}$

Schaltungen:



Multiplexer (MUX): selektiert durchzulassenden Eingang auf einen Ausgang, **Demultiplexer:** selektiert Ausgang auf den einzelner Eingang ausgegeben werden soll (invers);

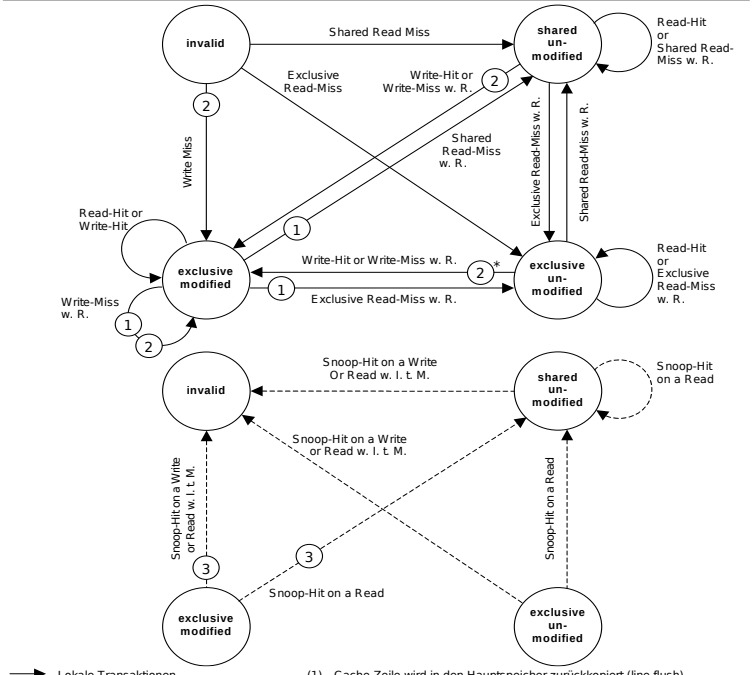
Zustandsautomaten: *Mealey-Automat:* $\mathcal{A} = (Q, \Sigma, \omega, \delta, \lambda, q_0, F)$, ($Q$ Zustände (endlich) mit q_0 Startzustand, F Endzustände, Σ endl. Alphabet, $\delta : Q \times \Sigma \to Q$ Übergangsfunktion, $\lambda : Q \times \Sigma \to \Omega$ Ausgabefunktion), *Moore-Automat:* wie Mealey-Automat, nur $\lambda : Q \to \Omega$. **Sequentieller Automat:** Mit Zuständen. **Kombinatorischer Automat:** Ohne Zustände.

Cache. **Lokalität:** *Zeitl. & räuml. Lokaliktät* (d.h. Daten, auf die gerade erst zugegriffen wurde/ deren Adresse nahe bei solchen liegt, im Cache halten - Wahrscheinlkt. für weitere Zugriffe hoch), Anwendung: räuml. Lok.: ganze Blöcke geladen, zeitl. Lok.: Verdrängungsstrategien; **Aufbau:** feste Anzahl von Cache-Einträgen, *Cache-Eintrag:* Adress-Tag → Cache-Line (Tag's werden gematcht, Cache-Lines beinhalten Daten), **Adressaufteilung:** $\boxed{\text{Tag} \mid \text{Index} \mid \text{Wordadresse} \mid \text{Byteadresse}}$ (Länge 16/32/64 Bit je nach Adressraum/System); **Berechnung:** #Cache-Lines = $k \cdot$ #Indexbits (für k-fach assoziativen Cache), Cache-Speicher = #Cache-Lines $\cdot$ $2^{\#\text{Wortbits}+\#\text{Bytebits}}$; **Assoziativität** k**:** Anzahl der Sätze (direct-mapped (DM): $k = 1$, vollassoziativ (FA) $k =$ #Cache-Lines, kein Index), *Vor/Nachteile:* FA - viele (parallel arbeitende) Vergleicher benötigt, weniger Verdrängung → hohe Effizienz & Hardwareaufwand, DM - weniger Vergleicher benötigt, mehr Verdrängung - geringere Effizienz & Hardwareaufwand; **Strategien: Schreib-Hit:** *write-back:* bei `WRITE` wird Block zunächst im Cache abgelegt, Zurückschreiben in HS beim Ersetzen der Cache-Line, *dirty bit* wird beim Schreiben gesetzt - zeigt an, ob HS + Cache übereinstimmen, *write-through:* direktes Zurückschreiben in HS, benötigt *write buffer*, um Daten zw. Cache & HS zu Puffern; **Schreib-Miss:** *write-allocate:* zu schreibender Block wird in Cache geholt + modifiziert (wird meist mit *write-back* kombiniert), *non-write-allocate/write-around:* Block wird nicht in Cache geladen; **Verdrängungstrategien:** *FIFO* (first in first out - zuerst in den Cache geladener Eintrag → zuerst ersetzt), *LRU* (least recently used - am längsten ungenutzter Eintrag → zuerst ersetzt), *LFU* (least frequently used - am wenigsten genutzer Eintrag → zuerst ersetzt, 1-2bits zu Speicherung der Häufigkeit), *CLOCK/Round-Robbin*, *optimal*; **MESI-Protokoll:** *Ziel:* Gewährleistung von *Datenkohärenz* bei MIMD-Systemen

MESI-Protokoll zur Erhaltung der Cache-Kohärenz

Das MESI-Protokoll ordnet jeder Cache-Zeile einen der folgenden vier Zustände zu:

Exclusive Modified (M):	Die Zeile befindet sich exklusiv in diesem Cache und wurde modifiziert.
Exclusive Unmodified (E):	Die Zeile befindet sich exklusiv in diesem Cache und wurde nicht modifiziert.
Shared Unmodified (S):	Die Zeile befindet sich noch in einem anderen Cache und wurde nicht modifiziert.
Invalid (I):	Die Zeile ist ungültig.



Endianness. **Big Endian:** MSB (most significant bit zuerst), d.h. Wort ergibt sich direkt aus linearem Hintereinanderschreiben des Speicherinhaltes; **Little Endian:** Bytes des Wortes müssen invertiert werden; **Einordung verschiedener Architekturen:** *Little Endian:* Intel x86-64 (I*ntel convention*), 6502, Z80, MCS-48, DEC Alpha, Altera Nios II, PDP-11; *Big Endian:* Motorola 6800 und 68k-Serie (*Motorola convention*), IBM POWER, System/360-370 etc.

Taxonomie von Rachnerarchitektur nach Giloi. *Tabelle:*

Operationsprinzip	Hardwarestruktur
→ Informationsstruktur	→ Hardwarebetriebsmittel
→ Steuerstruktur	→ Kooperationsregeln
	→ Verbindungstruktur

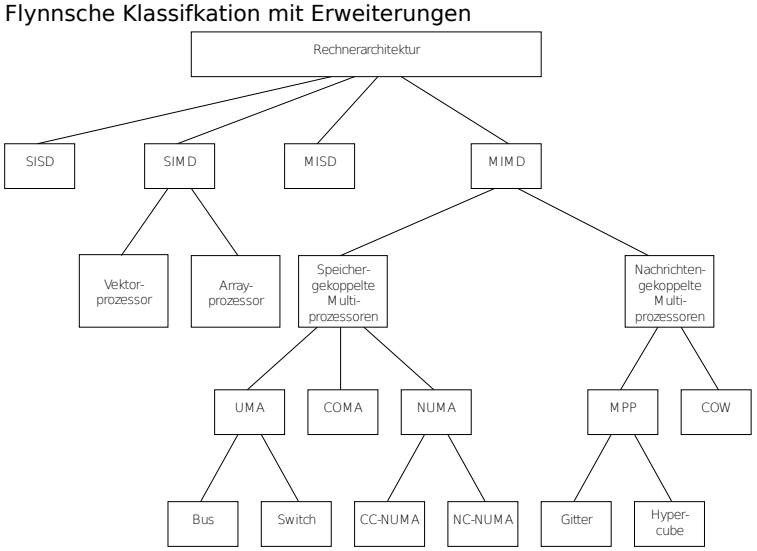
Weitere Unterteilung: **Informationsstr.:** a) Klassen von Datentypen (z.B.: Word, Feld, Stack, Liste etc.), b) Menge der Maschinendarstellungen (z.B: IEEE754, 2er-Komplement), c) Menge von Fkt.; **Steuerstr.:** a) Ablaufsteuerung (PC-getrieben (Befehlszähler-getrieben), datengetrieben, anforderungsgetrieben; Datenzugriffssteuerung (Adresslogik, einfache Wertzuweisung, assoziativer Zugriff)); **Hardwarebetriebsmittel:** a) Prozessorstruktur (Verarbeitungseinheiten auf Core, Zusammensetzung mehrerer Cores), b) Speicherstruktur (auch Register); **Verbindungstruktur:** (z.B.: Bussysteme, Single-/Multicore-System, Verbindungsnetzwerke); **Unterschied zu Brooks:** mehr Berücksichtigung der Interna (auf Nutzerseite); **Beispieleinordnung:** *Steuerwerk*: RA → HWstr. → HWbetr.m. → Prozessorstruktur; *Register*: RA → HWstr. → HWbetr.m. → Speicherstruktur; *Speicherbus:* RA → HWstr. → Verbindungsstruktur; *Festkommaformat:* RA → Op.pr. → Inf.str. → Menge der Maschinendarst. & Datenobjekte; *doppelt verkettete Liste:* RA → Op.pr. → Inf.str. → Klassen von Datenobj. → Strukturdatentypen; *Cache:* RA → HW.str → HW.b.m → Speicherstruktur; *PC-getriebene Ablaufsteuerung:* RA → Op.pr. → Steuerstr. → Ablaufsteuerung; *Gleitkommaformat:* RA → Op.pr. → Inf.str. → Menge der Maschinendarst. & Datenobjekte; *Assemblercode:* RA → Op.pr. → Inf.str. → Menge von Fkt.; *Verbindungsnetzwerk:* RA → HW.str. → Verbindungsstr.;

Brooks Definition (60/70er Jahre). interne Struktur & Organisation des Rechners vor Nutzer verborgen, RA= Interfacebeschreibung/Programmierschnittstelle (Befehlssatz, Speicherstruktur, Adressierungsmodi, Registerstuktur, Unterbrechungsbehandlung, E/A-Fkt. etc.)

Moore's Law. Verdopplung der Transistoren auf Chips pro 10 Monate (Erhöhung der Prozessorleistung, Energieaufnahme $\sim f^2$ → Taktfrequenz seit 2006 nicht mehr gesteigert, Lücke zwischen Speicher- & CPU-Performance wächst);

Flynnsche Klassifikation. **SISD:** Von-Neumann- & Harvard-Architektur, Single-Core-PC; **SIMD:** Vektor- & Feldrechner; **MISD:** leere Klasse; **MIMD:** Multicoresysteme; (S. . . *single*, M. . . *multiple*, I. . . *instruction*, D. . . *data*);

Flynnsche Klassifkation mit Erweiterungen



Speichergekoppelte Multiprozessoren. **Symmetrische Multiprozessoren (SMP).** Gleichartige Prozessoren über Bus/Kreuzschienenschalter/mehrstufiges Netzwerk verbunden; **Distributed-shared-memory-Systeme (DSM):** einheitl. Adressraum, aber Speicher physikalisch auf einzelne Verarbeitungsknoten verteilt; **UMA-Multiprozessoren (uniform memory acces):** alle Prozessoren greifen gleichermaßen af gemeinsamen Speicher zu (gleiche Zugriffszeit); jeder Prozessor kann lokalen Cache besitzen, Nutzen des Snooping-Bus-Verfahrens (Cache-kohärent), auch SMP genannt; *Beispiele:* Sun Enterprise 10000, **NUMA (non-uniform memery access):** Zugriffszeiten auf gemeinsamen Speicher variieren nach Ort der Speicherzelle (durch Verbindungsnetzwerk induziert); Speichermodule physikalisch auf Verarbeitungsknoten verteilt; gemeinsamer Adressraum; prozessorlokale Caches; *Beispiele:* Connection Machine CM-5 von TMC, **CC-NUMA (cache coherent NUMA):** Caches über gesamtes NUMA-Systems kohärent organisiert (directory-based coherence protocol); Datenbank über Befinden der einzelnen Cache-Zeilen & Zustand; *Beispiele:* Sequent NUMA-Q (4 Pentium Prozessoren mit L1-/L2-Cache, bis zu 4GB RAM), AMD-Mehrprozessorsysteme auf OPTERON-Basis, SGI-Systeme mit NUMAlink, früher: basierend auf Alpha-Prozessor EV7 (Digital Equipment Corporation), MIPS-R1x000-Prozessoren (SGI-Origin-Serie); **NCC-NUMA (non-cache-coherent NUMA):** NUMA ohne Cache-Kohärenz; lokale Speicherzugriffe (über Caches) und entfernte Zugriffe (vorbei an Caches); *Beispiele:* CRAY T3E (Vektorrechner); **COMA (cache-only mememory architecture):** Spezialfall des CC-NUMA; physikalsch verteilte Speichermodule ausschließlich Caches; gemeinsamer Adressraum; Speicher zieht benötigten Speicherblöcke an (*attraction memery* Prinzip); *Beispiele:* KSR 1, KSR 2 (Kandall Square Research);

MMX, SSE, AVX. **MMX (multi media extension - 1996):** große Register (64bit) werden in kleinere Register aufgeteilt, die parallel mit einem Befehl verarbeitet werden können (*PackedByte* (8×8bit), *PackedWord* (4×16bit) , *PackedDoubleWord* (2×32bit), *QuadWord* (1×64bit)); SIMD (Feldrechnerprinzip); Erweiterung für IA-32-Prozessorarchitektur; für Integer-Datenpakete **SSE (streaming SIMD extension - 1999):** große Register (128bit) in z.B. 4×32bit Register aufgeteilt (parallel verarbeitend); SIMD (Feldrechnerprinzip); Befehlssatzerweiterung für `x86`-Prozessorarch., speziell für Floating-Point-Datentypen; 8-16 Register, die mit `xmmi` (`i` variabel) beschriftet sind) **AVX (advanced vector extensions - 2008):** Vergrößerung der SIMD Register auf 256bit; neues 2-Operandenformat $a := b+c$, dadurch wird Quellregister nicht zerstört (SSE-Befehle $a := a+b$ im Zweioperandenformat); Sandy-Bridge realisiert als erstes AVX2;

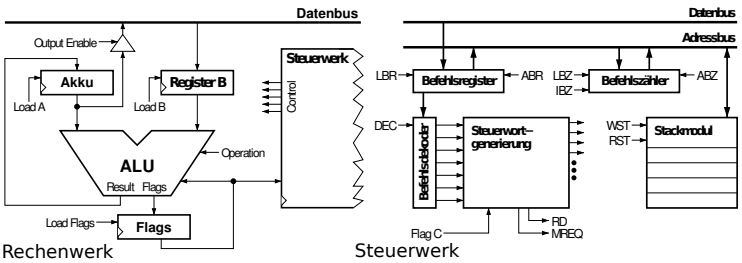
Speicherhierarchie (Bsp.daten). 1. **Register** (512, direkt), 2. textbfCaches L1 (32 kByte, 3 Takte), L2 (256 kByte, 8 Takte), L3 (8 MByte, 15 Takte), 3. **RAM/Hauptspeicher** (4/8 GByte, 100/120 Takte), 4. **Festplatte** (1TByte, Mio. Takte);

Adressierung. **Aufbau Befehlswort:**

Operationscode | Operanden | Adressteil

Adressierungsarten. Immediate Operand: Direkt-Operanden-Adressierung (d.h. ein Register wird für Operanden nicht gebraucht), Operand als Konstante im Befehl (z.B.: Assembler (#) `ADD R1,#17` oder `MOVE.B R3, #225`); `Immediate Adress:` effektive Adresse steht als absolute Adresse im Befehl, Operand im Speicher; **Register Direct:** Adresse steht als kurze Registeradresse im Befehl, Operand steht im Register (z.B.: `MOVE.W SP,R0` - transportiert Inhalt von `R0` in Stackpointerregister); **Register Indirect:** effektive Adresse steht im Register, Operand im Speicher (z.B.: Assembler (()) `MOVE.H R1,(R0)` - transportiert 16bit-Operanden, dessen Adresse in `R0` liegt nach R1); **Register Indirect & Displacement:** im Register stehende Adresse wirkt als Basisadresse, Offset/Displacement steht im Befehl und muss zu ihr addiert werden; Operand im Speicher (z.B.: `MOVE.H R2 #4(R3)`) **Scaled Index Addressierung:** wie Register Indirect mit Displacement nur dass zusätzlich noch scale $*$ index zur Adresse addiert wird; *scale* muss 2er Potenz sein (dann wird Multiplikation, die nicht möglich ist, zu Bitshifting); **Implizite Adressierung:** es wird implizit ein Register für den Opcode verwendet (z.B.: `ADD R5` bei Addition mit Akkumulator und Rückschreiben auf diesen, oder `ADD R3 R4` (2-Operandenformat, bei dem auf ersten Op. geschrieben wird); **Überdeckte Adressierung:** Ergebnis stimmt mit einem Operanden überein (z.B.: `ADD R1 R2`); ***n*-Adress-Format:** typische Operandenanzahl bei `ADD`, `SUB`, `AND` und `OR`; (z.B.: bei 0-Adress-Maschine - beide Operanden auf Stack in ALU, 1-Adress-Format - ACCU als impliziter, überdeckter Operand; 2-3 Adressformat - GPR (general purpose register));

Von-Neumann-Architektur. **Komponenten:** CPU, Systembus, Hauptspeicher, E/A-Einheit; *CPU:* beinhaltet CU/Steuerwerk (Befehlszähler, Befehlsregister, Befehlsdekoder, Steuer- & Statusregister/Stackmodul, zentrale Steuerschleife), Rechenwerk (ALU, Akkumulator - log./arithm. Op., Interaktion mit Steuerwerk durch arithm. Flags); *HS:* Daten & Befehle, gebildet durch linear adressierte Von-Neumann-Variablen (IS+value), zugriff auf gemeinsamen Bus; *Bus:* Adress-, Daten-, Steuerbus, für Befehle & Daten, Einschr. der Parallelität; **Befehlsabarbeitung (Von-Neumann-Zyklus):** 0. Befehlsadresse im Befehlszähler, 1. IF (instruction fetch), 2. ID (instruction decode)(3. OF (operand fetch)) 3. EX (execute), 4. WB (write back); *Schemata:*



Nachteile: Daten & Befehle haben gemeinsamen Bus/Speicher; sequentielles Arbeiten (ein Kontrollfluss/ Von-Neumann-Zyklus (In-Order-Execution)); CPU-Speicher-Geschwindingkeitsunterschied; **Lösung:** Paralellisierung (Pipelining, Multicore, Multithreading, superskalare Befehlsabarbeitung), out-of-order-execution, Caching, getrennter L1-Cache für Befehle und Daten;

Beispiel: Arbeitsweise Steuerwerk. **Generelle Abarbeitung von Befehlen (hier z.B.: `JC` mit Bedingungsflag C):** 1. Befehlswort anfordern, (d.h. Befehlszähler auf Adressbus legen `ABZ`, HS anfordern `MREQ`, HS lesen `RD`), 2. Wartetakt & Befehlszähler inkrementieren `IBZ`, 3. Befehlswort vom Speicher (Datebus) in Register laden (`LBR`), 4. wenn C, Adressteil des Befehlsregisters auf Adressbus (`ABR`), Befehlszähler vom Adressbus laden (`LBZ`); (4. Schritt bei `CALL` wäre z.B.: Befehlszähler auf Adressbus ausgeben (`ABZ`) und auf Stack legen (Rücksprungadresse) (`WST`), Adressteil des Befehlsregisters auf Adressbus ausgeben `ABR`) und in BZ laden (`LBR`);

Harvardarchitektur. **Unterschied zu VN-Arch.:** Befehle & Daten in unterschiedl. Speichern & benutzen unterschiedl. Busse; **Anwendung:** L1-Caches, Pipeline-Prozessoren; **Vorteil:** Parallelität beim Laden von Befehl+Operand, kein Gegenseitiges Verdrängen (Cache)/Überschreiben möglich;

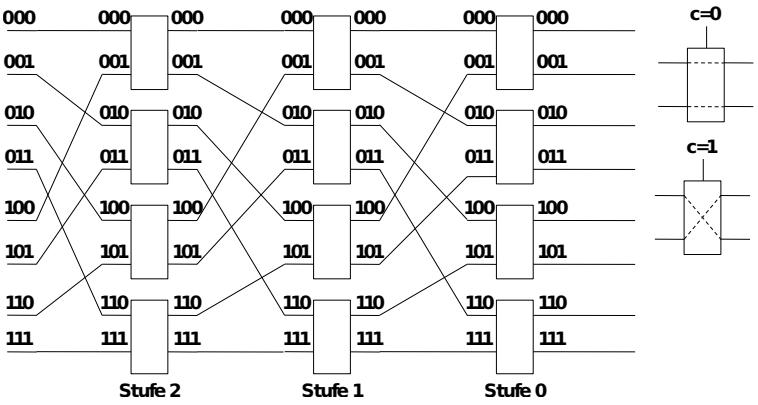
RISC/CISC: **RISC:** wenig Befehlsformate fester Länge, `LOAD/STORE`-Architektur, viele Register (bis zu 256), direkte Verdrahtung der Befehle im Dekoder (kein Mikrocode → schnellere Ausführung), *Vorteile:* schnelleres Dekodieren; weniger Hardwareaufwand; Sparsamkeit bei Adressierung, wenn $R0$ fest auf 0 verdrahtet;

Verbindungsnetzwerke. **statisch:** jeder Knoten hat feste Leitungen zu Nachbarknoten (Links), Netzwerk beschränkt sich auf Verbindungsleitungen, Verbindungs- & Vermittlungsfunktion nicht Bestandteil; **dynamisch:** Verbindungen schaltbar, konkrete Verbingung erst zu Kommunikationszeitpkt. vorhanden; **Kenngrößen:** *Grad* $d(v)$, *Durchmesser* d, *mittl. Abstand* $\overline{d}$, *Halbierungsbreite* b (minimal zu löschende Kantenanzahl, damit Netzwerk in 2 gleichgroße Teile zerfällt), *kleinste Erweiterung* e, *Einbettung* (i. gaphentheoretischen Sinn), *Konnektivität* $k := \min\{k_V, k_E\}$, *Knotenvernetzung* k_V (minimal zu löschende Knotenzahl, damit Netzwerk in 2 Teile zerfällt), *Kantenkonnektivität* k_E (analog für Kanten); **Entwurfsziele:** kleiner, konst, Grad, einfach skalierbar, kleiner Durchmesser, hohe Konnektivität, viele unabh. Pfade zw. 2 Knoten, hohe Halbierungsbreite; **Speziell für High Performance Computing (HPC)/Cluster Computing:** wenige Schaltzellen, gutes Blockierungsverhalten, geringe Latenz, hohe Übertragungsrate;

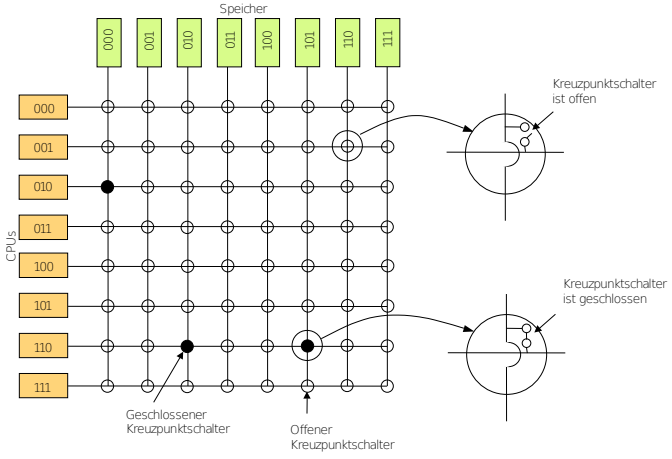
Bsp.: $n\times n$-

	$d(v)$	d	k	b	e
Ring C_n	2	$\lfloor n/2 \rfloor$	2	2	1
Graph K_n	$n-1$	1	$n-1$	$\lfloor n^2/4 \rfloor$	1
Stern S_n	1,$n-1$	2	2	$\lfloor n/2 \rfloor$	1
Binärb. B_n	1,2,3	$2\log_2(\frac{n+1}{2})$	1	1	$n+1$
2D Torus T_n	4	$2\lfloor n/2 \rfloor$	4	$2(\sqrt{n}+[1])$	$2\sqrt{n}+1$
H.-cube H_n	$\log_2 n$	$\log_2 n$	$\log_2 n$	$n/2$	n
rD Gitt. $G_{r,n}$	$r,\ldots,2r$	$r(\sqrt[r]{n}-1)$	r	-	-

Omega-Netzwerk: **Aufbau:** 2×2-Crossschalter (Betazellen) in $\log_2 n$ Stufen angeordnet, jede Stufe enthält $n/2$ Schalter, 'perfect shuffeling'; **Destination Routing:** Quell- & Zieladresse implizieren Schalterstellung (*destination tag*) durch **XOR-Routing:** Quelladresse $\oplus$ Zeiladresse $= c_{\log_2 n-1}\ldots c_0$ ergibt die nötigen Schalterstellungen (Schalter i-ter Stufe: $c_i = 0$ ungekreuzt ('straight'), $c = 1$ vertauschen ('exchange')); **Kollision:** tritt auf, wenn Quelle$_1$Ziel$_1$ mit Quelle$_2$Ziel$_2$ in mindestens $\log_2 n$ bits übereinstimmen;



UMA-Multiprozessorsystem mit Kreuzschinenschaltern



Paketvermittlung. Nachricht in Pakete aufgeteilt; Pakete werden unabh. voneinander durch Netzwerk transportiert; **Paket:** besteht aus *Header* (Routing- & Kontrollinformation), *Datenteil* (Anteil der Gesamtnachricht), *Endstück* (*trailer*, enthält Fehlerkontrollcode); **Arten: Store-and-Forward:** gesamtes Paket in jedem Zwischenknoten komplett gespeichert vor Weiterleitung; *Vorteil:* schnelle Freigabe von Verbindungen; *Nachteil:* schlechte Latenzzeiten, große Puffer je Knoten; **Virtual-Cut-Through:** Pakete in *phits* (physical units) unterteil, pipeline-artig durch Netz transportiert; an jedem Zwischenknoten wird nur Header betrachtet; bei freier Verbingung → Weiterleitung; bei Blockierung → Sammeln im letzten Erreichbaren Knoten; Vorteil: Freigabe des bisherigen Weges durch Puffern des Pakets im Knotenpuffer; Nachteil: große Puffer (entartet bei Block. zu Store-and-Forward); **Wormhole-Switching:** wie Virtual-Cut-Through, aber bei Blockierung bleiben alle phits an iherer Position; *Vorteil:* kleine Puffer, bessere Latenzzeiten; *Nachteil:* bisheriger Weg wird evtl. blockiert; **Leitungsvermittlung:** Leitung wird stationar aufgebaut und bleibt für gesamte Übertragungsdauer; *Nachteil:* Knoten werden blockiert, obwohl schon alle phits passiert haben, ineffizient für HPC (*high performance computing*); *Vorteil:* hohe Nettoübertragungsrate (keine Zeilcodierungen mitgeführt), hohe Leitungsstabilität;

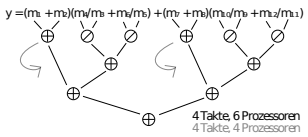
Parallelisierung: **Kennwerte (nach RA I):** CPI - Taktzyklen pro Befehl; IPC - Instruktionen pro Zyklus; t Arbeitszeit; T Periodendauer Takt; f Taktfrequenz; **Berechnung (nach RA I):**

Laufzeit $t = \#\text{Befehle} \cdot CPI \cdot T = \frac{\#\text{Befehle} \cdot T}{IPC}$

Vergleich: sequentieller vs. hypothetischer Parallelrechner. Gegeben: bestimmte Berechnungsvorschrift, testen auf Parallelisierbarkeit; **Kennwerte (RA II):** *Größen:* Z_p (Anzahl der Instruktionen bei Algo. mit p Prozessoren, Z_1 heißt seriell), T_p (Laufzeit bei p Prozessoren in Takten)

Speed-Up	$S_p = T_1/T_p$
Effizienz	$E_p = S_p/p$
Operationsredundanz	$R_p = Z_p/Z_1$
Auslastung	$U_p = \frac{Z_p}{pT_p}$
Effektivität	$F_p = \frac{S_p E_p}{T_1}$

Beispiel:



Pipelining. **Beschreibung:** gestaffelt paralleles Abarbeiten von Befehlen (IF-, ID-, EX-, WB-Stufen von Befehlen gestaffelt nebeneinander laufen lassen); **Berechnung der Effizienzsteigerung (nach RA I):**

Laufzeit (seriell)	$t_{\text{Pipeline}} = (\#\text{Pipeline-Stufen} + \#\text{Befehle} - 1) \cdot T_{\text{Takt}}$
Laufzeit (Pipeline)	$t_{\text{seriell}} = \#\text{Pipeline-Stufen} \cdot \#\text{Befehle} \cdot T_{\text{Takt}}$;
Speed-Up	$SP = t_{\text{seriell}}/t_{\text{Pipeline}}$
Effizienz	$EF = SP/S = \frac{\#\text{Befehle}}{\#\text{Befehle}+\#\text{Pipelinestufen}-1}$
Asympt. Werte	$EF \uparrow 1$, $SP \uparrow S$, $CPI \downarrow 1$ (für $\#\text{Befehle} \to \infty$)
Effektive Werte	$SP < \#\text{Pipelinestufen}$, $EF < 1$, $CPI > 1$

Out-of-Order-Execution. bei superskaleren Prozessoren (d.h. mehrere Funktionseinheiten wie ALU, FPU, Lade-&-Speichereinheit, Vektoreinheiten,; hohe Befehlsparallelität ($IPC > 1$ - IPC-fach superskalar)); **Arbeitsweise:** 1. IF (instruction fetch - Befehl laden), 2. IB (instruction buffer - Befehl in Warteschlange), 3. Warten des Befehls im Buffer auf Operanden (dann Verslassen des Buffers), 4. Befehl an passende Fkt.einheit Übergeben $ ausgeführt, 5. Ergebnis in Ergebniswarteschlange eingetragen (*register retirement/buffer*), 6. Nach Schreiben aller Ergebnisse früherereingetroffener, im Programmcode älterer Befehle → Schreiben des Resultats in Register; **Beispiel:** $f = 2,8$GHz, 3-fach superskalar ($IPC = 3$), 70ns Speicherlatenz ($\to 2,8 \cdot 70 = 196$ Takte): In der Zeit werden $3 \cdot 196 = 588$ Befehle abgearbeitet (d.h. potentiell 588 Bufferplätze → 128 wäre noch realistisch, da festverdrahtet, großer HWaufwand, Datenabhängigkeiten etc.);

Pipeline-/Out-of-Order-Execution-Konflikte (Hazards): **Datenhazard:** *read after write (RAW):* z.B.: `R1 = R2 + R3; R4 = R1+#1`, da erster Operand in 2. Anweisung eventuell noch nicht gestored wurde, müsste 2. Anweisung warten; 'Shortcuts' im Datenweg der Pipeline helfen; *write after read (WAR):* z.B.: `R1 = R2+R3; R2 = #2;`, Schreiben könnte vor Lesen stattfinden; tritt bei Pipeline eher nicht auf; *write after write (WAW):* z.B.: `R1 = R2+R3; R1 = #2`, falsche Schreibreihenfolge führt zu falschem Ergebnis; **Steuer/Kontrollhazards:** treten bei Instruktionen auf, die Befehlszähler verändern (z.B.: `JMC`), *Lösungen:* Sprungvorhersage (zusätzliche Hardwareeineheit, die Sprungwahrscheinl. berechnet); Delayed Branching (in der Zeit, in der Sprungziel ermittelt wird, werden andere unabh. Berechnungen durchgeführt), **Strukturhanzards:** mehrere Pipelinestufen griefen auf gleiche Ressource (z.B.: Quellregister) zu; *Lösungen:* Shortcuts innerhalb der Pipeline/ Anhalten der Pipeline (NOP);

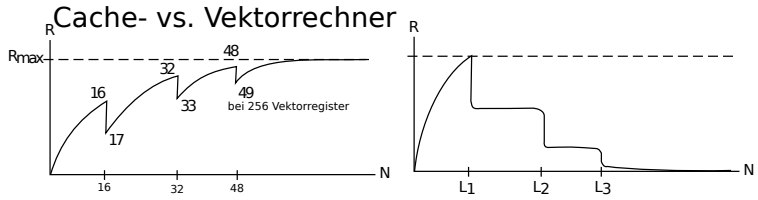
Leistungsbewertung. **Kenngrößen: IPS:** instructions per second; **IOPS:** integer operations per second; **FLOPS:** floating point operations per second; **IOOPS:** I/O operations per second (`READ, WRITE, RANDOM` etc.), nur ein Kern & keine Spezialregister benutzt; **Theoretische Peak-Performance:** theoretische maximal Erreichbarer Durchsatz eines Systems mit mehreren Kernen & Spezialregistern (z.B.: FPPP - floating point peak performance); **Beispielrechnung:** $f = 330$MHz, $IPC = 4$ (superskalar/Pipeline), $IOPC = 2$, $FLOPC = 2$; Dann: $IPS = IPC \cdot f = 1,32$GIPS, $IOPS = f \cdot IOPC = 660$MIOPS, maximale $IOOPS = f \cdot IOOPC = 330\text{MHz} \cdot 16 = 5,28$GIOOPS (Faktor 16: 4×1 IF, 4×1 WB, 4×2 OF

bei 2/3-Adressbefehl); Bei Akkumulator hier z.B. nur Faktor 2 (IF und OF, Akku kostet nichts);

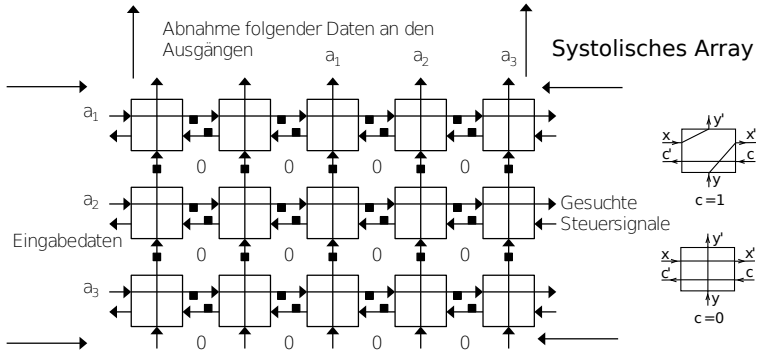
Vektorrechner. **Funktionsprinzipien:** *SIMD* (bei nur einem Vektorprozessor), viele *Speicherbänke* kein Cache, *Skalareinheit* für nicht vektorisierbare Probleme, mehrere Vektoreinheiten pro Vektorprozessor; meist mehrere Vektorprozessoren (damit als Ganzes MIMD), *arithmetisches Pipelining* (einzelne Phasen eines Befehls in einer Pipeline-Form organisiert, z.B.: FP-Add: 1. Exponenten vergleichen, Differenz, 2. Rechtsshift der Mantisse des kleineren Operanden, 3. Festkommaaddition der Mantissen, 4. Normalisierung), *Chaining* (Hintereinanderschalten von Pipelines, z.B.: bei $A = B. * C + D$ (elementweise Mult.) erste Addition, dann Multiplikation), nach Möglichkeit paralleler Zugriff auf HS;

Feldrechner. **Aufbau:** besteht aus N gleichartigen Einheiten (*processing elements* (PE)), die durch Verbindungsnetzwerk Daten austauschen können und eine gemeinsame Steuerung (*control unit* (CU)) haben); **Verarbeitungsprinzip:** *Pipelining*, *Chaining*; **Anwendung:** MMX, SSE, AVX;

Vergleich: Vektorrechner/Cache-basierend. **Vektorrechner.** maximale Performance für große Probleme; Einbruch, wenn Vektorregister voll ist & für einen einzelnen Wert ein fast leeres Register genutzt werden muss (wegen Laden des neuen Registers; k-faches der Registerlänge); bei $n \times n$-Matrix quadratisches Wachstum der Problemgröße in $n \rightarrow$ rascher Leistungsanstieg; **Cache-Rechner.** maximale Leistung for kleine Probleme für die L1-Cache reicht; Einbrüche immer beim Erreichen der nächst höheren Speicherhierarchie (L2/L3-Caches & Auslagerung auf RAM (*paging*)); *Beispiel Matrix-Multiplikation Floating-Point-Leistungskurven.*



Systolisches Array. **Definition:** zellulärer Automat, der aus gleichartigen, im Raum gleichmäßig angeordneten und gekoppelten Zellen besteht; Kopplungsmuster ist lokal; es tritt kein Broadcasting & Rippling auf; **Rippling:** Hindurchplätschern von Daten durch Zellen; **kein Rippling:** System getaktet; Freigabe des Signals zum Taktzeitpunkt; **Analogie zum Feldrechner:** siehe Feldrechner (PEs); **Analogie zum Vektorrechner:** Verallgemeinerung des Pipeline-Prinzips (1D - einfache Pipeline, 2D - Pipelines die sich in regulärer Weise beeinflussen); **Bedeutung:** Spezialrechner für Signal- und Bildverarbeitung;



ECS: $t_{\text{Rechnertyp}} = (k[*k'], d[*d'], w[*w'])$; $k \ldots$ Anzahl der nebenläufigen Steuerwerke, $k' \ldots$ Anzahl der spezialisierten Steuerwerke für Programm/Prozessorpipelining; $d \ldots$ Anzahl der nebenläufigen Rechenwerke (je Steuerwerk), $d' \ldots$ Anzahl der Pipelining Rechenwerke (je Steuerwerk), $w \ldots$ Anzahl der parallelen Bitstellen im Rechenwerk (z.B.: AVX: 256bit/ SSE 128/ MMX 64bit) ($W' \ldots$ Anzahl der elementaren Teilwerke);

HDN (hardware description notation) & DLX.

DLX: hypothetische Prozessorarchitektur mit RISC-Befehlssatz & 32 32bit-Registern (GPR) (Vorbild MIPS); Speicher ist byteadressiert

`R0` null; unveränderlich — `R1` reserviert für den Assembler
`R2-R3` Funktionsrückgabewerte — `R4-R7` Funktionsparameter
`R8-R15` beliebig — `R16-R23` Registervariablen
`R24-R25` beliebig — `R26-R27` reserviert für das Betriebssystem
`R28` Globaler Pointer — `R29` Stackpointer
`R30` Registervariable — `R31` Rücksprungadresse

Befehlsformate: *Register-Befehlsformat (R)*, *Immediate-Befehlsformat (I)*, *Jump-Befehlsformat (J)*:

	109876	54321	09876	54321	09876	543210
R	000000	Rs1	Rs2	Rd	unused	opcode
I	opcode	Rs1	Rd	immediate (IR)		
J	opcode	value (val)				

Tabelle:

Instr.	Description		Operation
ADD(I)(U)	add (i) (u)	R/I	$Rd \leftarrow Rs_1 + \text{IU}$
AND(I)	and (i)	R/I	$Rd \leftarrow Rs_1 \& \text{I}$
BEQZ	branch if $= 0$	I	$PC \leftarrow PC + ext(\text{IR})$ if $Rs_1 = 0$
BNEZ	branch if $\neq 0$	I	$PC \leftarrow PC + ext(\text{IR})$ if $Rs_1 \neq 0$
J	jump	J	$PC \leftarrow PC + ext(\text{val})$
JAL	jump and link	J	$R_{31} \leftarrow PC + 4; PC \leftarrow PC + ext(\text{val})$
JALR	jump and link reg.	I	$R_{31} \leftarrow PC + 4; PC \leftarrow Rs_1$
JR	jump register	I	$PC \leftarrow Rs_1$
LHI	load high bits	I	$Rd \leftarrow \text{IR} << 16$
LW	load word	I	$Rd \leftarrow_{32} \text{M}[Rs_1 + ext(\text{IR})]_{0..31}$
OR(I)	or	R/I	$Rd \leftarrow Rs_1 \mid \text{I}$
SEQ(I)	set if $=$ to (i)	R/I	$Rd \leftarrow (Rs_1 = ext(\text{I})?1 : 0)$
SLE(I)	set if $\leq$ (i)	R/I	$Rd \leftarrow (Rs_1 \leq ext(\text{I})?1 : 0)$
SLL(I)	shift l. logic. (i)	R/I	$Rd \leftarrow Rs_1 << (\text{I}\%8)$
SLT(I)	set if $<$ than (i)	R/I	$Rd \leftarrow (Rs_1 < ext(\text{I})?1 : 0)$
SNE(I)	set if $\neq$ to (i)	R/I	$Rd \leftarrow (Rs_1 \neq ext(\text{I})?1 : 0)$
SRA(I)	shift r. arithm. (i)	R/I	$Rd \leftarrow Rs_1 >>_a (\text{I}\%8)$
SRL(I)	shift r. logic. (i)	R/I	$Rd \leftarrow Rs_1 >> (\text{I}\%8)$
SUB(I)(U)	subtract (i) (u)	R/I	$Rd \leftarrow Rs_1 - Rs_2$
SW	store word	I	$\text{M}[Rs_1 + ext(\text{IR})]_{0..31} \leftarrow_{32} Rd$
XOR(I)	exclusive or (i)	R/I	$Rd \leftarrow Rs_1 \hat{} \text{I}$

Transfer	$\leftarrow$
Speicher	M
Trans. Länge	$\leftarrow_n$
Einzelbit	X_n
Bitkette	$X_{n..m}$
Wiederholen	X^m
Verketten	$\#\#$
L. Shift	$<<$
R. Shift	$>>$
R. Shift arith.	$>>_a$
+, -, etc. wie in C	

$ext(a) := (a_{16})^{16} \# \# a_{16..31}$
$\text{IU} := (\neg U \; ? \; ext \circ)(I \; ? \; \text{IR} : Rs_2)$, $\text{im} := I \; ? \; \text{IR} : Rs_2$

Beispiele:

DLX	HDN
`ADD R2, R3, R4`	`R2 ← R3+r4`
`ORI R5, R0, 0x102`	`R5 ← 0x102`
`LW R3, 0(R5)`	`R3 ← M[R5+0]##M[R5+1]##M[R5+2]##M[R5+3]`
`SUBI R2 R0, 5`	`R2 ← -5`
`loop: ADDI R2, R2, 1`	`R2 ← R2+1`
`SW 0(R3), R4`	`M[R3+0] ← R4`
`BNEZ R2, loop`	`n ←` PC+$(IR_{16})^{16}$##$IR_{16...31}$`; if(R2!=0) PC ← n`

weitere (laden eines Bytes aus Speicher und 0/vorzeichenerweiterte Zuordnung)

DLX	HDN
`LBU Rd, 0(Rs1)`	`Rd ←` 0^{24}`##M[Rs1+`$(IR_{16})^{16}$`##`$IR_{16...31}$`]`
`LB Rd, 0(Rs1)`	`Rd ←` $(M[Rs1 + (IR_{16})^{16} \# \# IR_{16...31}])_0^{24} \# \# M[\ldots]$